Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach

Robin Bussenot, Hervé Leblanc and Christian Percebois
Institut de Recherche en Informatique de Toulouse
118 Route de Narbonne, 31400 Toulouse, France
robin.bussenot@irit.fr, leblanc@irit.fr, percebois@irit.fr

***Abstract*—An airplane is composed by many complexes and embedded systems. During the integration testing phase, the design office produces requirements of the targeted system, and the test center produces concrete test procedures to be executed on a test bench. In this context, integration tests are mostly written in natural language and manually executed step by step by a tester. In order to formalize integration tests procedures dedicated to each system with domain specific languages approved by testers, and in order to automatize integration tests, we have introduced agile practices in the integration testing phase. We have chosen a Behavior Driven Development (BDD) approach to orchestrate Domain Specific Test Languages produced for the ACOVAS FUI project.**

I. Introduction

An airplane is composed by many complexes and embedded systems. Due to the criticality of such systems and constraints of verification and validation, a process of certification is necessary in order to commercialize them. In this context, the system integration testing phase is an important topic of the certification process. As a System Under Test (SUT) is composed by hardware and software, a dedicated test bench is needed to interact with it. We focus our work on integration test procedures.

In most cases, integration tests are written in natural language and are executed on a test bench step by step manually. The lack of formalism of the natural language introduces misinterpretations. The step by step execution can be fastidious, expensive and error prone. Tests written in natural language and executed manually are more focused on how the test has to be performed rather than the elicitation of the feature to be tested. The test intention is lost because it is mixed with implementation details. When test procedures are transformed into some scripting languages (Python, Lua, XML, C ...) executed on a test bench, the gap between a test scenario and the corresponding script is too large and does not permit debug. Testers have domain skills on the kind of system to test and not necessary computer knowledge to understand test scripts. The lack of test automation implies the waste of time during test replays. The lack of formalization implies the loss of intention in test procedures and then the lack of reusing for other avionic programs.

Our problematic is to improve the formalization of test procedures by introducing several best practices coming from software engineering and the efficiency of the integration testing activities. We used capabilities of Domain Specific Languages (DSLs) to formalize specific test languages dedicated to each system of an airplane. From these languages, program transformations are designed to generate executable code for some kind of test benches with their script languages. BDD promotes test cases naturally described in languages that focuses on preoccupations of a specific domain. The business value of test cases is improved because tests are centered on their intent and can be shared by all the stakeholders of the system under test development. Our work was supported by a FUI[1] project named ACOVAS aiming at introducing agile methods with a new generation of test benches.

After presenting system integration testing goals, basis, and needs (Section II), we introduce Domain Specific Languages and Behavior Driven Development, the two main concepts of our proposition (Section III). Then we present our behavior driven development framework for system integration testing (Section IV). The proof of concept is detailed by the two cases studies we have implemented during the project (Section V). We position our work (Section VI) and we conclude and give some perspectives (Section VII).

II. System integration testing

The process development of an entire avionic system is more complex than a software development process because an avionic system is composed by many systems and each system is composed by hardware and software. This process follows several V-cycles, and the test management process belongs to all activities of the process. Then, we recall the objectives of our proposition from observations of the ACOVAS partners.

A. V-cycles

An avionic system has several specific characteristics: real time, safety critical, embedded, and fault tolerant. The development process of each of them follows a V-model. An embedded system merges hardware and software with their specific V-cycle too. These two V-models are named the W-model. The resulting system of these two processes is integrated itself into a main process that follows a V-model too.

Two kinds of testing activities coexist: classic testing in isolation and In-the-Loop Testing. The latter provides models and

[1]Fonds Unique Interministériel (FUI) is a French program dedicated to support applied research, to help the development of new products and services susceptible to be marketed in short or middle term.

simulators to put the SUT in simulated flight conditions. This kind of testing allows integration testing as soon as possible. More precisely, testing is also performed since requirements level thanks to Model-In-the-Loop testing (MiL). The SUT behavior and the external environment are simulated through models to ensure that requirements are correct. In Software-In-the-Loop testing (SiL), the real software is tested with an emulated hardware. In Hardware-In-the-Loop testing (HiL), the real software is integrated into the target hardware. In-the-Loop testing responds to reactive and real time constraints of embedded systems. This kind of testing needs environment models to simulate collaborations between the SUT and other systems.

B. Test procedures management

Test procedures management has a dedicated process too, as explained by Sommerville [1] for a Software Engineering point of view. In System Engineering, this process depends on companies. In the ACOVAS context, the design office and the test center are the two stakeholders that manage integration test procedures. The design office produces requirements of the target system to develop, while the test center produces concrete test procedures to be executed on a test bench to verify that the system conforms to its requirements.

More precisely, the design office produces a test plan providing the integration strategy and acceptance criteria refined from system requirements. A Lab Test Request (LTR) is derived from a test plan. It is focused on test objectives and is related to a specific version of the SUT. A test objective corresponds to one expected feature of the SUT. It is described by a single sentence completed by a flight scenario which is independent of test means and an environment model from which results of a test can be exploited. The test center produces a test strategy that matches with test objectives of the LTR. Test strategy includes test means needed to perform test cases. Test procedures are refined from test cases descriptions according to test means. Generally, test procedures are stored in textual documents.

C. Needs

Actually, many test procedures are written in natural language. We identify two different ways to use test procedures written in natural language:

1) Test procedures are read and interpreted by a tester that directly executes each instruction step by step on a test bench.
2) Test procedures are translated *via* ad hoc transformations into a dedicated scripting language that allows automatic execution.

The manual intervention of a tester for the comprehension or the translation of a test procedure written in natural language is fastidious and error prone. More precisely, the list below enumerates the main preoccupations of the ACOVAS stakeholders:

- improve tests automation,
- reduce the time of tests implementation,
- manage changes in tests configuration,
- identify test solutions from the design phase,
- introduce specific languages for procedures definitions.

A framework dedicated to support the design and the implementation of test procedures will help testers to produce formalized tests. This formalization allows to compile a test procedure into some scripting languages and then increases the automation of tests. The formalization has been implemented using a DSL workbench.

Moreover, we would like to give an holistic approach [2] by focusing on the communication means provided by test procedures. Our goal is not to build a complete automatic tool chain but to help testers, developers and designers to achieved their tasks in a collaborative manner. Our solution is a framework supporting a BDD approach.

III. Background

A. Domain Specific Languages

Each system of an aircraft requires specific domain skills. The ATA 100 (Air Transport Association of America) [3] standard contains the reference to the numbering system which is a common standard for all commercial aircraft documentation. It contains some chapters dedicated to systems that compose an aircraft (Aircraft general, Airframe systems, Structure, Propeller/Rotor, Power plant). This commonality allows greater ease of learning and understanding for pilots, aircraft maintenance technicians, and engineers alike. Testers must be specialized on each system and it is difficult for them to gain competencies both on the SUT and programming languages. The goal of a tester is to focus on domain specifications and not to design complex test procedures. This point requires the introduction of small computer languages understandable by testers and that produce executable procedures on a test bench.

A Domain Specific Language (DSL) is a computer language that allows to provide a solution for a particular class of problems [4]. We can cite for instance SQL, initially named SEQUEL (Structured English Query Language), which was designed to manipulate and retrieve data stored in relational databases. Other examples are HTML and XML configuration languages. A DSL clearly focuses on a small domain in order to be easier to learn by domain experts than general purpose languages (GPLs).

Producing a DSL brings many advantages [5]:

- makes easier expressing domain concerns,
- increases the common knowledge about a domain,
- improves team communication,
- can be used by domain experts that not necessarily have computer programming knowledge,
- can be managed by specific tools as IDE for GPL languages,
- hides GPLs complexity,
- can generate many lines of code in GPL from few lines.

Our specific testing languages focus on the description of test scenarii for ATA 21 and ATA 42. Each DSL addresses a particular domain and are designed for test engineers only.

We named Domain Specific Test Language (DSTL) this new category of DSL [6].

B. Projectional vs textual editors

There are many tools to manage and help the implementation of new languages [7]. The most used are Xtext [8], Spoofax [9] and Meta-Programming System (MPS) [10]. All these tools are doing almost the same thing but the main difference is the way of editing a program. Editors can be projectional as they are in MPS, or textual as usual in Xtext and Spoofax. According to Martin Folwer in [4]: "Projectional editing thus usually displays a wider range of editing environments - including graphical and tabular structures - rather than just a textual form."

With a projectional editor, the end-user is guided by the structure of the language and no syntax errors are allowed. With a textual editor, the end-user has an empty text frame and must know the grammar of the language to produce test scenarii. Syntax errors are allowed and require corrections by the user himself.

From the point of view of a DSTL programmer, generators in MPS come with a list of features that allow to go through Abstract Syntax Trees (AST) [11]. Rules programmers are only focused on the semantic of transformations and do not take care about syntactical problems. Textual editors provide small stub generators and are model-to-text generation oriented. For those reasons, we chose MPS to conduct our experimentation.

C. Behavior Driven Development

BDD [12] has emerged to reduce the gap between unit tests and the elicitation of the specifications of the product's behavior. "BDD leads the development of features by designing functional tests used to validate those features" (Agile Alliance, Glossary BDD).

BDD combines Test Driven Development (TDD) and Domain Driven Design (DDD) principles to encompass the wider picture of agile analysis and automated acceptance testing for software production. BDD argues that the expected behavior of software should be described in testable scenarii related to functional requirements. Those testable scenarii can be seen as isolated tests in Acceptance Test Driven Development (ATDD) with a business view in order to improve communication means.

Tests in BDD are structured following a Given-When-Then canvas [13], corresponding to the Gherkin language, which is the simplest way of describing a behavior: given a nominal state, when an event changes this nominal state, then the system should produce the expected response. This canvas helps testers to discover and describe a scenario. Scenarii, corresponding to unit tests are used as specifications, and also have in mind to produce automated tests easily understandable by domain experts.

Another advantage of BDD is the communication improvement between stakeholders by using the vocabulary of the business domain and understandable notations by the whole development team. BDD provides a common view of the domain and helps developers and test specialists to understand domain experts needs.

In our approach, a DSTL encapsulates vocabulary and specific actions for testing and managing a kind of SUT as promoted by the BDD approach by using the Gherkin language. In teams adopting a BDD style of development, specialists of the domain write acceptance tests in a formalized language near a natural language. Programmers complete acceptance tests with glue code that provides links with the real software under test. In our case, we have designed a pivot language that a test programmer can use to link acceptance tests to the parameters of the SUT and to the capabilities of the test bench.

IV. A DSTL WORKBENCH

Our first idea is to use DSLs to design a specific test language dedicated for each kind of avionic domain (ATA). All domains involved in the development process of an airplane are covered by a family of such languages that we named DSTLs. Our second idea is to orchestrate three levels of testing languages (languages dedicated to testers, a unique language for test programmers, and scripting languages for test benches) in a BDD approach. The Fig. 1 shows those three levels of abstraction. In this section, we explain how we merge BDD concepts with our DSTLs.

A. The big picture explained

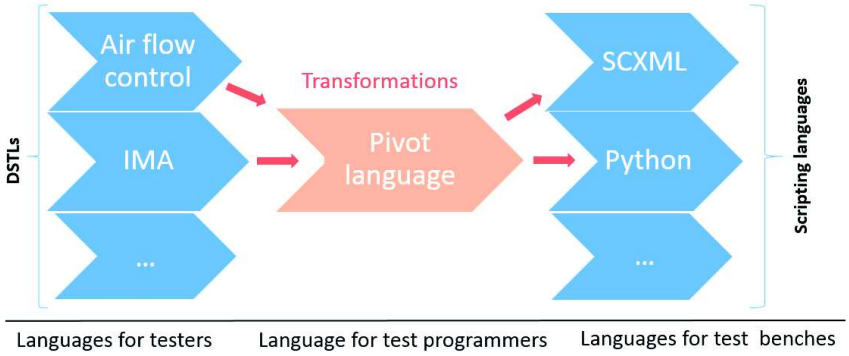
State Chart XML (SCXML) and Python with Data Distribution Services are two examples of executable languages on a test bench. The pivot language has been designed to reduce the semantic gap between languages for testers and languages for test benches. This language is programming oriented and serves to give a readable view of the executable code and to reuse transformation rules to several target languages. Finally, some business specific testing languages are proposed to testers for each kind of system.

Nowadays, the process to design those languages is empirical. To add a new DSTL, programmers have to design projectional editors, semantic constraints and an AST-to-AST transformation into the pivot language. To add a new target language, programmers have to develop an AST-to-text transformation based on transformation patterns used by other transformations proposed by the framework.

B. DSTLs and Gherkin language

DSTLs are testers oriented. Each language is dedicated to a specific domain to take into account testers practices and testing constraints. As example of testing domain we can quote network features, air flow control, flight commands, *etc*. We describe in the next section two experiments on DSTLs, one for ATA 21 (Air flow control) and one for ATA 42 (IMA). Gherkin language patterns can be used to explain the intention of each unit test. However, for the integration testing phase of reactive systems, tests must follow a flight plan that focus on a unique test objective. The Given-When-Then pattern is transposed into a flight plan pattern.

Fig. 1. The big picture



C. DSTLs and BDD approach

High level test languages are designed to replace test procedures in natural language. Since those languages use a common vocabulary of a specific domain, they can be considered as ubiquitous languages. Those languages are an efficient way of communication between testers, testers and designers, and also testers and test programmers. "BDD offers more precise guidance on organizing the conversation between developers, testers and domain experts" (Agile Alliance, Glossary BDD). Moreover, as BDD extends TDD and ATDD approaches, a test must focus on a single aspect of a program. Tests describe a unique intention to which the code has to conform. During system integration testing phase, test cases must focus on a unique test objective.

D. The pivot language

The semantic gap between high level testing languages and scripting languages for test benches is too wide. On the one hand, high level languages must be closed to natural languages. On the other hand, scripting languages are similar to assembly languages enhanced by the management of interactions with the test bench. The pivot language is a simple imperative language that serves to map high level statements to executable code. It is composed by a set of four atomic statements: `set`, `check during`, `check until` and `call` *externalTool* that cover about 80% of basic actions required by tests. The `set` statement of the pivot language allows to assign a value to an avionic parameter. The goal of the `check until` statement is to ensure that a state can be reached until an amount of time given by the statement. The `check during` statement ensures that a state of the SUT is stable during an amount of time given by the statement. This language allows to compose test suites, test cases and unit tests as a xUnit framework. It is open to the addition of new statements and minimizes the effort to translate test procedures into another scripting language. It is test programmer oriented, however it could be used by testers to directly encode their test procedures.

E. Expected benefits

The first benefit is that test procedures can be automated depending on the capabilities of test benches and supported scripting languages at our disposal. Test procedures keep a human readable form nearest of old test procedures written in natural language and executed manually. They are able to explicit a unique intention by test case. As in TDD approaches, tests are isolated from each other and a specific error message can be thrown when a test failed or when a test cannot be run due to a problem with the SUT. The debugging phase during new test campaigns is simplified. The second benefit is the separation between preoccupations of testers (only focusing on expected behavior explained) and programmers (only focusing on glue code to improve the automation of tests) as it is promoted by the BDD approach. The last benefit is the reuse of transformation patterns for the generation of executable code from a unique pivot language.

V. Cases Studies

The global specification of an airplane is refined into system domains, systems, sub-systems, and finally equipments. These systems are regrouped in the ATA classification in more than hundred chapters [3].

Two kinds of test procedures are provided by the ACOVAS project:

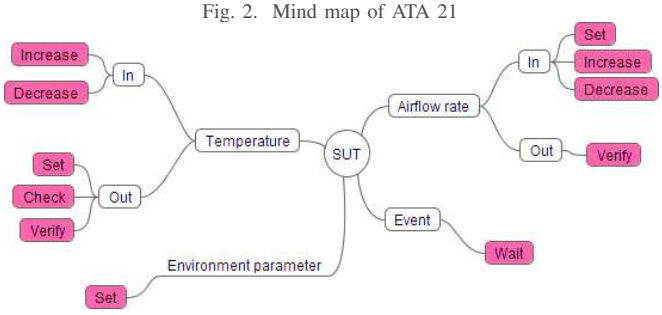
1) Airflow control test procedures (ATA 21) provided by an equipment manufacturer are textual descriptions in an intermediate level language that will be transformed manually into Excel spreadsheets. These spreadsheets are automatically transformed into an XML dialect.
2) Integrated Modular Avionic (IMA) test procedures (ATA 42) provided by an aircraft integrator are textual descriptions in natural language. Due to the wide expressiveness of natural languages, a similar procedure written by two different testers may have significant differences. Moreover, these procedures will be manually executed by another tester.

A. ATA 21

The ATA 21 focuses on airflow control systems of an airplane. Those systems manage the temperature of each area by controlling the airflow rate of each duct supplying them.

The mind map presented by the Fig. 2 has been retro-conceived closely with experts of the ATA 21 from procedures written in pivot language. It shows the main

Fig. 2. Mind map of ATA 21



concepts needed to formalize a test case in this domain: temperature, airflow rate, event and environment parameter. Each path from `SUT` to a leaf is represented by a DSTL statement. For example, the `Increase` statement can be instantiated in two manners, the first one to increase the temperature of an area (`Increase the temperature of Expected_Temperature_Area1 up to 20.°c`) and the second one to increase the air flow rate of a set of ducts (`Increase the air flow rate of Ducts_Area1 up to 0.2 kg/s`).

The generator coming with the DSTL translates each statement into the pivot language. Generally, variables manipulated by DSTL statements correspond to a set of real avionic parameters. Statements used to decrease or increase a temperature or an airflow rate are translated into a collection of `set` statements of the pivot language. Statements to verify a behavior or to wait an event are translated into a single `check until` statement of the pivot language. The statement to check the temperature of an area is translated into a single `check during` statement of the pivot language.

We implemented our conceptual framework for an air flow control test procedure. The DSTL procedure contains 48 lines of code, and the generated one into the pivot language contains 68 lines. We generated 2010 lines of SCXML code and 290 lines of Python code.

B. ATA 42

The ATA 42 focuses on Integrated Modular Avionic systems. Those systems are used as data concentrator and provide data to other systems of the airplane.

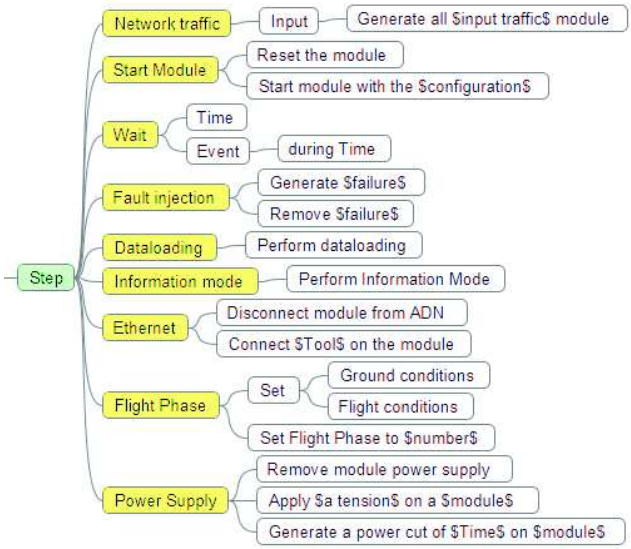
Procedures dedicated to ATA 42 provided by ACOVAS partners are described thanks to natural language. Procedures are composed by a set of test cases and test cases are completed by sentences divided in five kinds (`Step`, `Check`, `Trace`, `Log` and `Reminder`). We have studied 10 procedures containing around 3700 tagged sentences. We have used Natural Language Processing tools [14] to provide statistics about verbs the most used for each kind of sentence. The table I shows the number of sentences and the rate of each kind of sentence.

We have collaborated with experts of ATA 42 to produce the mind map of all needs for this domain formalization. We have designed seventeen kinds of `Step` statements, one for `Log` statements, one for `Check` statements and five for `Trace` statements. The Fig. 3 presents the mind map for the `Step` statements that formalize interactions with the SUT.

TABLE I
NUMBER OF SENTENCES SORTED BY KIND

	Step	Check	Log	Trace	Reminder
Number	1274	1047	594	520	269
%	34,4	28,3	16	14	7,2

Fig. 3. Mind map for `Step` statements



Testers have formalized their procedures thanks to our DSTL workbench successfully. To improve the formalization, some semantic contraints must be added to complete projectional editors. To assume test automation, a generator must be provided for `Step`, `Trace` and `Log` statements. To provide an oracle, a new formalization and a new generator must be provided for `Check` statements.

VI. Related Work

We considered some contributions related to the transposition of best practices coming from software engineering into avionic systems testing. We identify three scientific axes:

1) Test language ([15]),
2) Test automation in an avionic context ([16], [17]),
3) Agility in an avionic context ([18], [19], [20], [21], [22]).

A. Test language

The European Telecommunications Standards Institute (ETSI) has standardized a general purpose language named Test Description Language (TDL) [15] dedicated to the specification of test descriptions and the presentation of test execution results. This language is between test purposes described in natural language and the necessary complexity to implement tests in Test Control Notation version 3 (TTCN-3) for example.

Unfortunately, a tester without computer programming skills cannot understand TDL test descriptions. None of the partners of the ACOVAS project have adopted this standard which comes from the telecommunications industry.

B. Test automation in an avionic context

Ott [16] focuses on new testing activities arising from IMA architectures. These activities concern automated bare IMA modules and a network of configured IMA modules. The automation of these activities is based on generic test templates. In the first case, a user must instantiate data configurations for a bare IMA module. In the second case, data configurations are automatically generated from the ICD (Interface Control Document) of a network of IMA modules. In this work, the effort of automation uniquely concerns unit tests of IMA modules and networks of IMA modules. A more general framework is proposed by Guduvan et al. [17] who use a model-driven development approach to generate test cases for all kinds of avionic tests. The structure of test procedures is defined thanks to a dedicated meta-model. Test cases are components of a test suite or a test group. They must be decomposed into behavioral sequences of statements, and each statement corresponds to a specific test order. As in our work, the test procedure structure is domain independent, while specific statements are related to a specific domain.

C. Agility in an avionic context

The usefulness of Scrum and XP was studied by Salo et al. [18] in many embedded software development projects from European organizations. This study reveals that these organizations seem to be able to apply agile methods in their projects. A process named Safe Scrum was proposed by Stålhane et al. [23] to introduce agile principles in an embedded software certification context. A test first approach was proposed by Manhart et al. [19] for high-speed software engineering for embedded software. This work focuses on unit testing and mixes agile practices with conventional process activities. All these works are dedicated to software embedded development only.

A DSL is very closed to an ubiquitous language used by the BDD approach [13]. The structure of an ubiquitous language comes from the business model and contains terms which will be used to define the behavior of a system. The main idea is that customers and developers share the same language without ambiguity.

VII. Conclusion

Integration test procedures are conceived to verify several behaviors expected of a system responding to several stimuli during a flight plan. Usually, integration test procedures are described by a specific domain vocabulary. These procedures are good candidates to adopt a behavior driven approach. We have formalized and automatized test procedures for ATA 21 and two demonstrations have been made on two different test benches with StateChart XML and Python scripting languages.

We will design a new DSTL dedicated to another ATA to propose a global process to guide the design of new DSTLs. These future works will be supported by the ESTET (Early Systems TEsTing) project leaded by the DGAC[2].

Acknowledgment

We would like to thank the ACOVAS (outil Agile pour la COnception et la VAlidation Système) partners for their collaboration and their involvement during the project.

References

[1] I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[2] L. Von Bertalanffy, *General system theory, Foundations, Development, Applications*. New York: George Braziller, 1968.

[3] s. S-tech Entreprises, "Ata100 and section headings," http://www.s-techent.com/ATA100.htm.

[4] M. Fowler, *Domain-Specific Languages*, ser. Addison-Wesley Signature Series (Fowler). [Online]. Available: https://books.google.gr/books?id=ri1muolw_YwC

[5] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. [Online]. Available: http://dslbook.org

[6] R. Bussenot, H. Leblanc, and C. Percebois, "A domain specific test language for systems integration," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops. New York, NY, USA: ACM, 2016, pp. 1–10, article 16. [Online]. Available: http://doi.acm.org/10.1145/2962695.2962711

[7] M. Fowler, "Language workbenches: The killer-app for domain specific languages," http://www.martinfowler.com/articles/languageWorkbench.html.

[8] Itemis, "XText," http://www.eclipse.org/Xtext/.

[9] E. Visser, "Spoofax," http://strategoxt.org/Spoofax.

[10] Jetbrains, "Meta-programming system," https://www.jetbrains.com/mps/.

[11] F. Campagne, *The MPS Language Workbench*. FABIEN CAMPAGNE, 2013-2014, vol. 1.

[12] D. North, "Introducing BDD," 2006, https://dannorth.net/introducing-bdd/.

[13] C. Solís and X. Wang, "A study of the characteristics of behaviour driven development," *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 383–387, 2011.

[14] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland: Association for Computational Linguistics, 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P14-5010

[15] ETSI. (2015) The Test Description Language (TDL); Part 1 : Abstract Syntax and Associated Semantics. ES 203 119-1, http://www.etsi.org/technologies-clusters/technologies/test-description-language.

[16] A. Ott, "System Testing in the Avionics Domain," Ph.D. dissertation, University of Bremen, 2007.

[17] A.-R. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber, "A Meta-Model for Tests of Avionics Embedded Systems," *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pp. 5–13, 2013. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004320000050013

[18] O. Salo and P. Abrahamsson, "Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum," *Software, IET*, vol. 2, no. 1, pp. 58–64, February 2008.

[19] P. Manhart and K. Schneider, "Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE, 2004, pp. 378–386.

[2]French Civil Aviation Authority

[20] M. R. Smith, A. K. C. Kwan, A. Martin, and J. Miller, "E-TDD - embedded test driven development a tool for hardware-software co-design projects," in *6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, 2005, pp. 145–153. [Online]. Available: http://dx.doi.org/10.1007/11499053_17

[21] M. Karlesky, W. Bereza, and C. Erickson, "Effective test driven development for embedded software," in *Proceedings of IEEE International Conference on Electro information Technology*, May 2006, pp. 382–387.

[22] A. Wils, S. Van Baelen, T. Holvoet, and K. De Vlaminck, "Agility in the avionics software world," in *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, ser. XP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 123–132. [Online]. Available: http://dx.doi.org/10.1007/11774129_13

[23] T. Stålhane, T. Myklebust, and G. Hanssen, "The application of safe scrum to IEC 61508 certifiable software," in *11th International Probabilistic Safety Assessment and Management Conference 2012, PSAM11 ESREL 2012*. Curran Associates, Inc., 2012, vol. 8, pp. 6052–6061. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84873160115{\&}partnerID=tZOtx3y1